Partitioning in DB2 Using the UNION ALL View

Calisto Zuzarte *Robert Neugebauer* *Natt Sutyanyong* *Xiaoyan Qian* *Rick Berger*
IBM Toronto *IBM Toronto* *IBM Toronto* *IBM Toronto* *IBM Boca Raton*
© Copyright International Business Machines Corporation 2002. All rights reserved.

Abstract

In today's relational databases, it is not uncommon to hear of terabyte-size databases. As it becomes necessary to store ever increasing volumes of data in a single table within the database, more people need to know how to manage this data. The solution to many of these situations is typically one of "divide and conquer." The commonly recommended solution when using DB2® Universal Database™ Version 7 on the various workstation platforms is to use a partitioned database. While it is well recognized that there needs to be other long term solutions, this paper discusses an existing "partitioning" solution—namely, the approach of using a UNION ALL view. The treatment of the UNION ALL view in the DB2 query rewrite component of the DB2 SQL compiler has been sufficiently enhanced to make it worth considering when there is a requirement to manage data that is large but needs to be viewed as a single relation.

Contents

1. Introduction....2
2. A business application and possible database Issues....2
3. Using a UNION ALL view....3
4. DB2 query rewrite and the UNION ALL view....5
4.1 Local predicate pushdown....6
4.2 Redundant branch elimination....7
4.3 Join pushdown....10
4.4 GROUP BY pushdown....13
4.5 The result of the query rewrite transformations....14
4.6 Runtime branch elimination with parameter markers....16
5. Benefits of UNION ALL views....18
5.1 Better control of maintenance window utilities....19
5.2 Easier to roll data in and out....19
5.3 Ability to leverage different storage media....20
5.4 More granular performance tuning....20
5.5 Easier to modify the schema and the data....20
5.6 Decreased I/O costs....22
5.7 Increased query parallelism....22
5.8 Optimizing UNION ALL Views in a federated environment....23
6. Limitations of using UNION ALL views....23
6.1 Risk of view materialization with complex queries....23
6.2 Problems guaranteeing uniqueness across tables in a view....23
6.3 Restrictions on inserting into a UNION ALL view....24
6.4 Limitations on the number of branches in a UNION ALL view....24
6.5 Increased compile time and memory usage....24
6.6. Future enhancements....25
7. Conclusion....25

1. Introduction

Business intelligence applications that are used today require that large amounts of historical data be stored. One common application is to store and analyze prior business transactions, such as sales data, over a period of several years. It is easy to envision a sales forecasting system that stores all sales transactions for three years with 500 MB of data generated daily. That would require active storage of approximately 500 GB of data just for sales.

Early versions of DB2 were limited to storing data in table spaces consisting of 4KB pages. With these 4KB pages, an individual row within a table uses a four-byte row identifier (RID) to locate a row. Of these four bytes, three are used to identify a page and one to identify the offset within a page. The maximum number of pages was therefore limited by the maximum integer that could be stored in 3 bytes. So with 16 million 4KB pages, the limit for a single table was 64 GB. Subsequent versions of DB2 introduced larger page sizes that allowed this limit to be stretched to 512 GB by using a 32KB page size table space. Yet, as the example in the previous paragraph showed, this limit was a problem. Other than not having enough room to define indexes and other tables, there was no room to grow.

To overcome these table size limits and to achieve scalability through parallelism, DB2 adopted a shared-nothing architecture in 1995. This partitioned database allowed the table to be partitioned on several nodes of a cluster or within a single SMP server where each partition had the table size limits. The size of the table could now be extended depending on how many partitions could be provided. The partition ID extended the RID to allow for much more data to be stored in the table. The data is distributed among various partitions using a hash partitioning scheme by hashing the values of one or more columns in the table. This is the general recommendation to overcome the size limits of a table in DB2.

There might be a situation when a single-partition DB2 user has not anticipated the growth of a table or does not want to move to a partitioned database in the near time frame. One approach that might be worth considering is instead of storing the data in a single table, use a UNION ALL view over multiple tables. Application queries can refer to this view to look at the data in all the component tables as a single entity. The purpose of this paper is to discuss the advantages and disadvantages of this approach.

This paper is organized as follows.

- Section 2 introduces a typical business application and possible database issues.
- Section 3 presents the approach of using a UNION ALL view to solve these related issues.
- Section 4 describes the work done by the query rewrite component of the SQL compiler in DB2 in order to optimize the query. Each type of optimization is explored in detail, laying out the evolution of a query. Finally, the optimized query is compared with the original query.
- Sections 5 and 6 describe the benefits and limitations of using a UNION ALL view, respectively.
- Section 7 is a conclusion.

2. A business application and possible database issues

A worldwide trading company has decided to create a data warehouse for its sales data. The finance department wants to track and analyze the sales revenue across geographies for all products sold on a periodic basis. The logical design of the tables is as follows.

```
sales(   sales_date   date not null,
    prod_id    integer,
    city_id    integer,
    channel_id   integer,
    revenue  decimal(20,2))
```

```
products(prod_id    integer,
    prod_desc  varchar(50),
    prod_group_id   integer,
    prod_group_desc varchar(50),
    launch_date   date,
    terminated   char(1))

geographies( region_id  integer,
    region     varchar(50),
    country_id   integer,
    country    varchar(50),
    state_id     char(3),
    state    varchar(50),
    city_id    integer,
    city     varchar(50))

channel( channel_id   integer,
    channel    varchar(50),
    channel_cost  decimal(20,2))
```

The *sales* table stores sales transactions over a period of three years. It is estimated that the sales transactions collected from all the sales worldwide can be as large as 500 MB daily. The *products* table records all products manufactured. The *geographies* table references a city_id to its corresponding city name, state, country, and region. The *channel* table refers to all the channels the company uses to sell its products and a consolidated channel cost.

With daily sales transactions of 500 MB, the *sales* table can grow to an approximate size of 15 GB in a month and 180 GB in a year. On a single-partition database, it will take just three years of data to reach the limits of the table. This could be a problem if, for whatever reason, moving to a partitioned database is inappropriate. The first problem is the ability to store such large amounts of data given the single partition limits required by this particular trading company.

Query performance on this table may be a concern. Indexes on the table could have more levels than those on a smaller table. If there are many probes of the index, the extra disk I/O to navigate through the index may not offer the best performance.

3. Using a UNION ALL view

Other than a multi-partitioned database, a practical approach to deal with the size of table and to manage administration tasks is to physically partition the sales table into a set of smaller tables. In particular, the sales table can be represented by tables of the same column definition but with each of the tables representing different period of the sales transactions. For example, we may have table *sales_0198* for sales transactions in January 1998, table *sales_0298* for transactions in February 1998 and so on. Then we “glue” all the tables together as a view named *all_sales* using the UNION ALL construct. We will refer to this kind of view as a *UNION ALL view*. Branches of the UNION ALL view do not need to have a uniform structure or range of data. This allows complete customization based on performance and hardware characteristics.

One way of distributing the data could be done as follows:

- Data for the oldest year can be put in a single base table.
- Data for each quarter of the middle year can be put into separate tables.
- Finally, a single base table can be created for each month of the current year.

The view can be named *sales* so that applications need not be changed.

In order to guarantee that the table sales_0198 will contain only sales transactions from January 1998, we need to put a check constraint in the definition of the table as follows. Check constraints ensure that the data integrity is maintained in accordance with the definition of the constraint.

```
create table sales_0198(
    sales_date   date not null,
    prod_id integer,
    city_id    integer,
    channel_id integer,
    revenue decimal(20,2),
    constraint   ck_date
        check
        (sales_date between '01-01-1998' and '31-01-1998'))
```

The check constraint is also necessary for DB2 query rewrite to improve the performance of the query against the *all_sales* view by ensuring that only the relevant monthly sales tables are accessed, as is described in more detail in Section 4.

Another option to achieve the same result is to define a WHERE clause on every table in the UNION ALL view. You can use this option if there is a screening process in place before data is loaded into the table to ensure that data is loaded to the proper table.

The following statement shows the definition of the view *all_sales*:

```
create view all_sales as
    (
    select * from sales_0198
    where sales_date between '01-01-1998' and '31-01-1998'
      union all
    select * from sales_0298
    where sales_date between '01-02-1998' and '28-02-1998'
      union all
    ...
      union all
    select * from sales_1200
    where sales_date between '01-12-2000' and '31-12-2000'
    );
```

The optional WHERE clauses (shown in bold for identification) are needed only if the base tables do not define check constraints for the date ranges of the sales transactions.

If you are familiar with Oracle's *partitioned view*, you may be wondering why Oracle plans to withdraw support of that feature. This feature in Oracle is based on the same principle of dividing the table, but it is more limited in the associated structures defining the view. All tables must have similar schema and indexes. It does not have the flexibility and independence associated with the basic UNION ALL approach. Many of the benefits that can be obtained using the UNION ALL view approach discussed here are not applicable to Oracle's *partitioned view*. Presumably, due to the availability of Oracle's *range partition* and limitations of Oracle's *partitioned view*, it is being phased out.

4. DB2 query rewrite and the UNION ALL view

The DB2 query rewrite component of the SQL compiler is a powerful transformation engine. The optimizations listed below are performed by the query rewrite component. This list of optimizations includes only those that are explicitly relevant to UNION ALL views; there are many other optimizations that will benefit most queries.

The DB2 query rewrite engine attempts to prune the number of tables that need to be accessed in the processing of the query. The following optimizations work together to improve the performance of a query over a UNION ALL view:

- Local predicate pushdown.
- Redundant branch elimination.
- Join pushdown.
- Group by pushdown.

For each of these optimizations, we describe

- The benefits of the optimization.
- The result of each optimization on a sample query.
- The measures that are in place to deal with any possible drawbacks.

Throughout Section 4, we show the evolution of a query for the business problem above.

Some information that a company might want to obtain is the total revenue per active product generated in each city by all distribution channels during January and February of 2000. You can express this as follows:

Query 1:

```
select s.prod_id, p.prod_desc, g.city, c.channel,
       sum(s.revenue) as “Total Revenue”
from products p, geographies g, channel c, all_sales s
where s.prod_id = p.prod_id and
  s.city_id = g.city_id and
  s.channel_id = c.channel_id and
  s.sales_date between ‘01-01-2000’ and ‘29-02-2000’ and
  P.terminated = ‘N’
group by s.prod_id, p.prod_desc, s.city_id, g.city, s.channel_id, c.channel
```

A graphical version of this query is depicted in Figure 1.

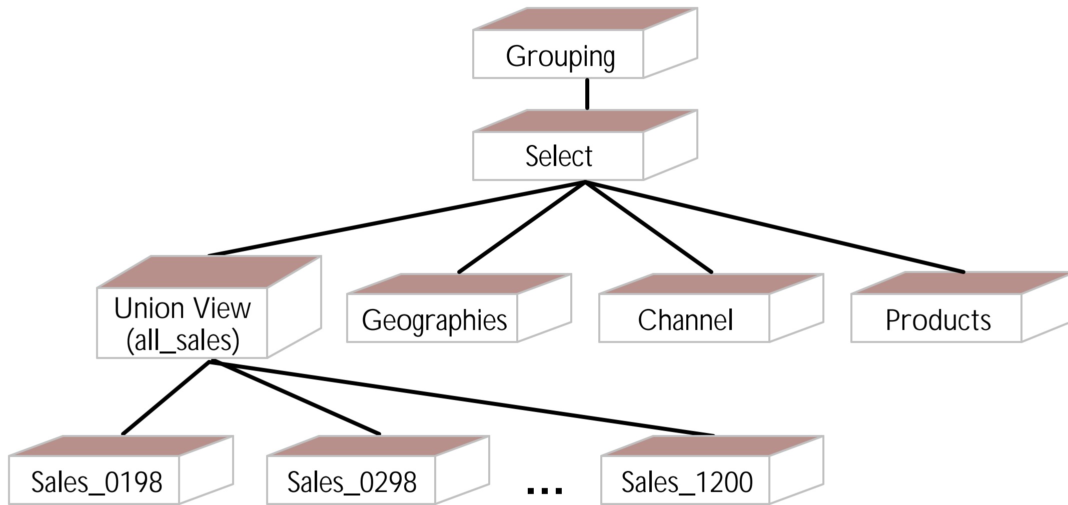

Figure 1. Graphical representation of Query 1

All of the optimization methods listed above can be applied to this query, and we will show in the following sections how this query is transformed into its final form.

4.1 Local predicate pushdown

The DB2 query rewrite component pushes eligible local predicates down through SELECT, join, UNION, or GROUP BY. The purpose of "predicate pushdown" is to apply the restrictions earlier to reduce the intermediate data flows between operations. If the local predicates (ie., predicates not involving other tables) can be pushed down to the operations at the lowest level when accessing the table, the restrictions made by those predicates then will eliminate any unqualified rows and feed only the qualified rows to the next upper level operations, and so on.

In the example query given in Section 4, there are three local predicates that are eligible to be pushed down: `'01-01-2000' <= s.sales_date`, `s.sales_date <= '29-02-2000'`, and `p.terminated = 'N'`. These two predicates that involve s.sales_date are pushed through the UNION ALL to each of the partitioned sales tables; the predicate that involves p.terminated is pushed to the products table.

After local predicate pushdown, the query looks like this :

Query 2:

```
with p1 as (select prod_id, prod_desc from products
    where terminated = 'N'),
  s1 as (select * from sales_0198
    where '01-01-2000' <= sales_date and sales_date <= '29-02-2000'),
  s2 as (select * from sales_0298
    where '01-01-2000' <= sales_date and sales_date <= '29-02-2000'),
  ...
  s36 as (select * from sales_1200
    where '01-01-2000' <= sales_date and sales_date <= '29-02-2000'),
  sales2 as(select * from s1
        union all
      select * from s2
        union all
```

```
        ...
        select * from s36)
  select s.prod_id, p.prod_desc, g.city, c.channel, sum(s.revenue) as "Total
Revenue"
  from p1 p, geographies g, channel c, sales2 s
  where s.prod_id = p.prod_id and
    s.city_id = g.city_id and
    s.channel_id = c.channel_id
  group by s.prod_id, p.prod_desc, s.city_id, g.city, s.channel_id, c.channel
```

The predicate is pushed down to all the base tables of the UNION ALL view *all_sales* as depicted in Figure 2.

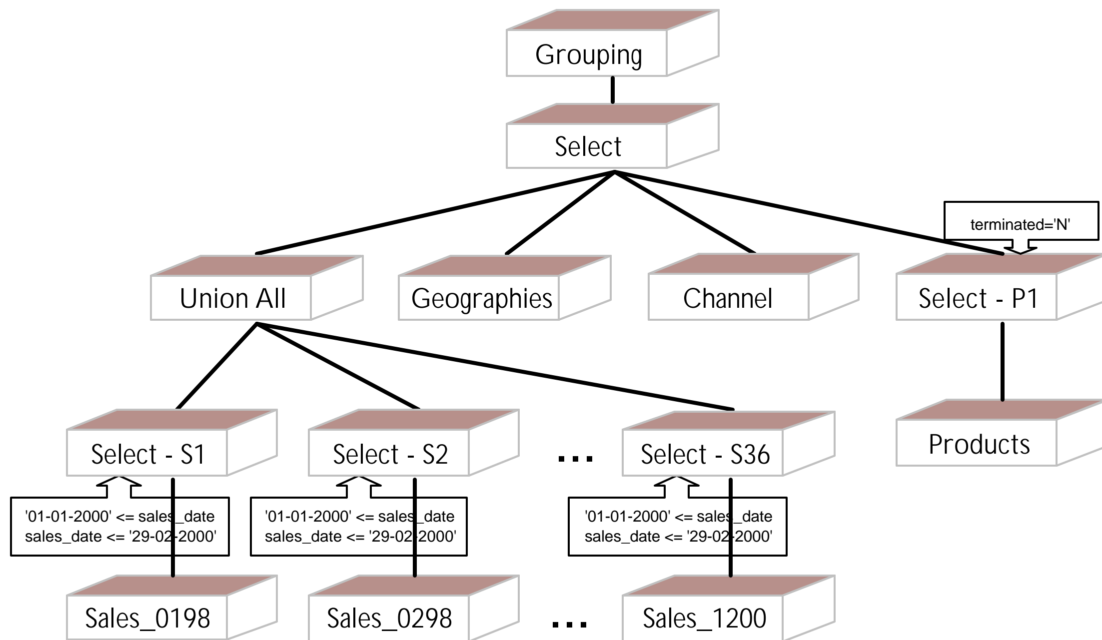

Figure 2. Graphical representation of Query 2

The benefit of applying predicates early is that the number of rows can be reduced earlier. In Query 2, we are now filtering the rows from the products table and the UNION ALL view *before* the join. Assume that the products table has 30,000 rows, but that only 1000 of them meet the condition `terminated='N'`. Any join that involves the products table will now be more efficient because there are fewer rows to join, and DB2 does not need to eliminate a substantial amount of rows from the result of the join.

4.2 Redundant branch elimination

This optimization method works in combination with local predicate pushdown to improve query performance. Redundant branch elimination works by detecting inconsistencies in the predicate set for each branch. If a given subset of the predicates is inconsistent, there is no way that the operation of that branch will return any rows. If this branch of the UNION ALL view is removed, it will not affect the result of the query.

Let us take a look at the created view S1. The predicates shown in bold are the check constraints defined on the base table *sales_0198*.

Query 3:

```
select * from sales_0198
where '01-01-2000' <= sales_date and
  sales_date <= '29-02-2000'
   and '01-01-1998' <= sales_date and sales_date <= '31-01-1998'
```

It is not difficult to prove that there are no rows that satisfy all the four predicates in the SQL statement above. Specifically, sales_date stored in table *sales_0198* cannot be smaller than 01-01-98 and simultaneously larger than 01-01-2000. When the DB2 optimizer detects this inconsistency, it knows that this branch of UNION ALL does not need to be accessed and can be dropped from the UNION ALL view even before executing the query. After eliminating the redundant branch, Query 2 in section 4.1 now looks like this:

Query 4:

```
with p1 as(select prod_id, prod_desc from products
    where terminated = 'N'),
  s25 as (select * from sales_0100
    where '01-01-2000' <= sales_date and sales_date <= '29-02-2000'),
  s26 as (select * from sales_0200
    where '01-01-2000' <= sales_date and sales_date <= '29-02-2000'),
  sales2 as(select * from s25
        union all
      select * from s26)
select s.prod_id, p.prod_desc, g.city, c.channel,
        sum(s.revenue) as "Total Revenue"
from p1 p, geographies g, channel c, sales2 s
where s.prod_id = p.prod_id and
  s.city_id = g.city_id and
  s.channel_id = c.channel_id
group by s.prod_id, p.prod_desc, s.city_id, g.city, s.channel_id, c.channel
```

This is shown graphically in Figure 3. As you can see, the number of branches in the UNION ALL view has been reduced from 36 to 2. There are now 34 fewer table or index accesses.

DB2 can detect inconsistencies most effectively with equality or inequality predicates (<, >, <=, >=, =, <>, between); however, DB2 can also detect inconsistencies with more complicated predicate types, including IN and OR predicates. With the more complicated predicate types, it might possibly be too difficult, too expensive, or just not possible for DB2 to detect inconsistencies.

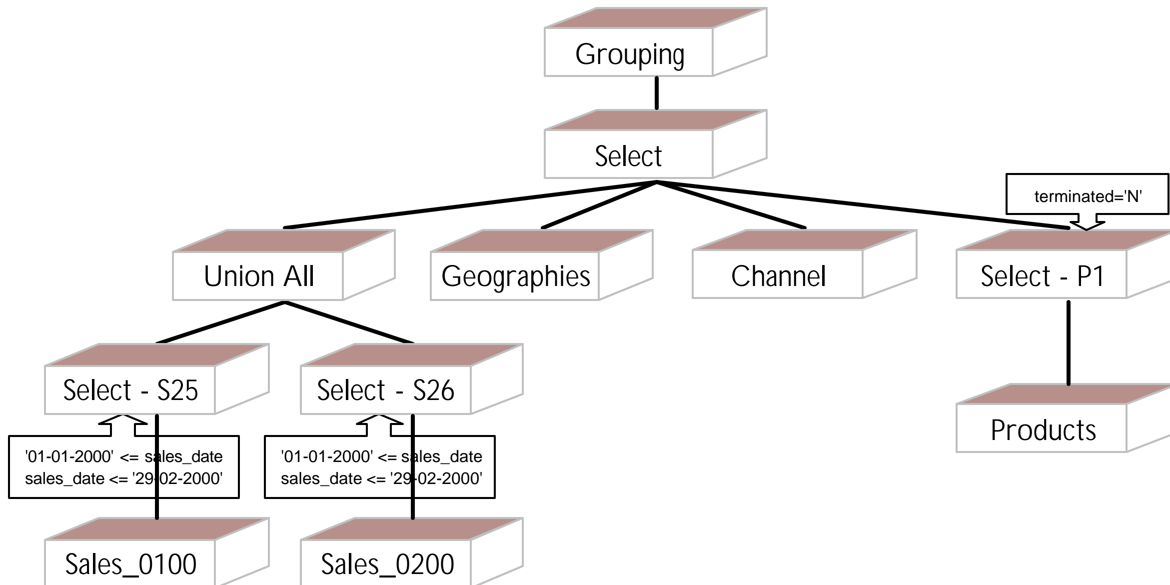

Figure 3. Graphical representation of Query 4

For example, if a query has multiple IN predicates, it requires comparing every element of each IN predicate. This comparison is expensive to do, and IN predicates would not be inconsistent in most cases. DB2 does make an exception when there is an equality predicate and an IN predicate. In that case, DB2 does do full comparisons to detect inconsistencies.

For example, assuming that there is a UNION ALL view *all_products* for the *products* table that is partitioned by the *prod_group_id* column. The view is set up so that each base table contains exactly one product group and is enforced by an equality check constraint on *prod_group_id*. With this in place, the following query is issued:

Query 5:

```
select * from all_product
where prod_group_id in (1, 3, 5)
```

DB2 can eliminate accesses to all base tables except the ones with prod_group_id as one of the elements in the IN predicate.

Predicates that involve a function, e.g., `UPPER(state) = 'ONTARIO'`, cannot be used to prove inconsistency in order to eliminate branches (for reasons other than the fact that Ontario is a province and not a state!). The exceptions are the YEAR and MONTH functions. For example, if the predicate `YEAR(sales_date)=2000` is specified, it would be converted to `'01-01-2000' <= sales_date and sales_date < '01-01-2001'`. Similarly, the predicates `YEAR(sales_date)=2000 and MONTH(sales_date)=2` can be converted to `'01-02-2000' <= sales_date and sales_date < '01-03-2000'`. Attempting to use an IN predicate or an OR predicate along with the date function will fail to enhance the pruning.

This is only an issue if the query predicates or check constraints use a function on the UNION ALL view partitioning column. A solution to this is to use a generated column as the partitioning column. For example, consider the table *geographies* that has a UNION ALL view defined over it using the *state* column as the partitioning column. Ordinarily, branch elimination would not occur because of the UPPER function in the

predicates. However, an uppercase representation of the *state* column could be generated and used as the partitioning column for the UNION ALL view, as shown below:

```
create table geographies_1(
      region_id  integer,
      region     varchar(50),
      country_id   integer,
      country    varchar(50),
      state_id     char(3),
      state    varchar(50),
      state_up generated always as (UPPER(state)),
      city_id    integer,
      city     varchar(50))
```

Query rewrite substitutes the predicate `UPPER(state) = 'ONTARIO'` with `state_up = 'ONTARIO'`, thus allowing branch elimination.

In DB2 Version 7, it is not always possible to remove branches from the access plan at compile time. There are some situations where DB2 query rewrite introduces special execution time predicates that it evaluates upfront to see if it needs to access a branch or not. This is not a bad thing and works well in some situations, such as when parameter markers are present.

4.3 Join pushdown

For a UNION ALL view, the DB2 SQL optimizer tries to perform the "join pushdown" to the base tables. Without pushing down the joins or the join predicates, DB2 would need to materialize the UNION ALL view and then do the join. Join pushdown ensures that any indexes on the base tables are available to make the join operation more efficient. This pushdown of joins usually has the same benefit as local predicate pushdown because it may reduce the number of rows flowing to upper operations. The join pushdown is limited to equi-join predicates only and when the number of the remaining branches in the UNION ALL view is less than 36. These limits will be revised upwards in future versions of DB2. Join pushdown is applied after any redundant branch elimination has occurred.

Let's return to the example from Section 4. After join pushdown, Query 4 looks like this:

Query 6:

```
with  s25 as (select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0100 s, products p, geographies g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    s26 as (select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0200 s, products p, geographies g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    sales2 as  (select * from s25
                union all
              select * from s26)
```

```
select prod_id, prod_desc, city, channel, sum(revenue) as "Total Revenue"
  from sales2
  group by prod_id, prod_desc, city_id, city, channel_id, channel
```

This is shown graphically in Figure 4.

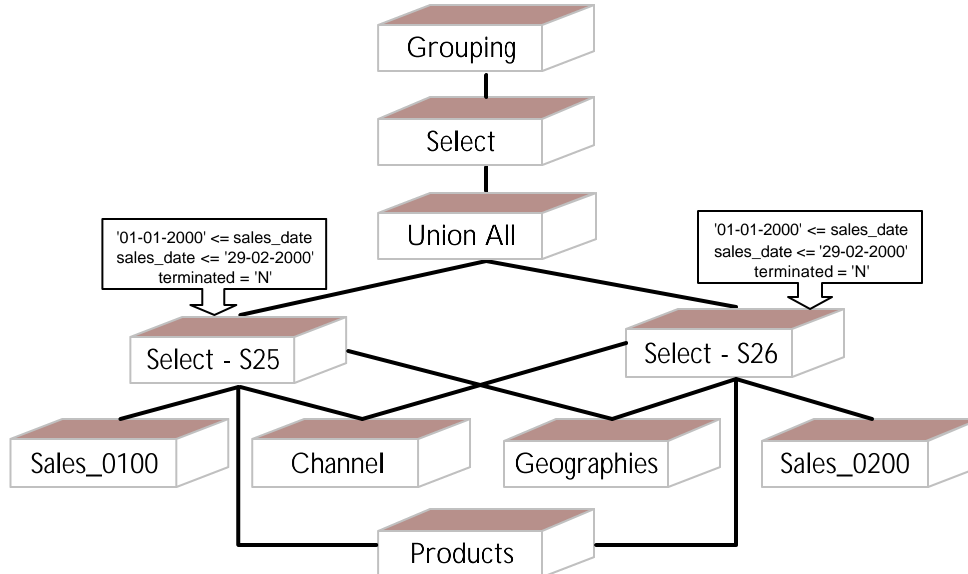

Figure 4. Graphical representation of Query 6 after join pushdown

The joins are now being done on each individual sales base table. This is more efficient for the following reasons:

- There are fewer rows in the *sales_0100* and *sales_0200* tables than in the *all_sales* UNION ALL view.
- There are a greater selection of join methods available because any indexes on the sales partition are now available to be used.

Join pushdown is usually a good optimization, but there are some drawbacks:

- Compile time.
- Memory required to replicate complex access plans for each UNION ALL branch.

With join pushdown, the compile time of a query may increase. This is especially true if the join structures that are pushed down are very complex. The number of joins in the example above has been doubled. That means the optimizer needs to determine the optimal join ordering for each branch of the UNION ALL view independently. In this simple example it would not be an issue, but in a large UNION ALL view, compile time could increase substantially if there are a large number of joins as a result of the pushdown.

The replication of the join structure to each branch of the UNION ALL view could also cause the memory requirements of the query to increase if the join structures that are pushed down are very complex. That is because each branch uses its own copy of the same join structure.

There are limits in place that prevent pushdown when the optimizer determines that the substructures are too complex. Additionally, outer join is not pushed down in current versions of DB2 even though in some situations, this is possible.

If there are more than one UNION ALL type views joined in a query, the optimizer also considers pushing down one UNION ALL view through the other. If the geographies table had been a UNION ALL view

partitioned by the *region* column, the result of join pushdown would have been very different. Query 7 shows what it would be like. For simplicity, the geographies UNION ALL view only has three branches.

Query 7:

```
with  s251 as(select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0100 s, products p, geographies_1 g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    s261 as(select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0200 s, products p, geographies_1 g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    s252 as(select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0100 s, products p, geographies_2 g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    s262 as(select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0200 s, products p, geographies_2 g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    s253 as(select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0100 s, products p, geographies_3 g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    s263 as(select s.prod_id, p.prod_desc, g.city, c.channel, s.revenue
      from sales_0200 s, products p, geographies_3 g, channel c
      where '01-01-2000' <= sales_date and sales_date <= '29-02-2000' and
        s.prod_id = p.prod_id and
        s.city_id = g.city_id and
        s.channel_id = c.channel_id and
        p. terminated = 'N'),
    g1 as(select * from s251
           union all
         select * from s261),
    g2 as (select * from s252
           union all
         select * from s262),
    g3 as(select * from s253
```

```
          union all
        select * from s263),
  sales2 as  (select * from g1
              union all
            select * from g2
              union all
            select * from g3)
select prod_id, prod_desc, city, channel, sum(revenue) as "Total Revenue"
from sales2
group by prod_id, prod_desc, city_id, city, channel_id, channel
```

The joins are now being performed below the unions. As before, each join has a smaller input set and is able to make use of indexes on the base tables.

Like regular table join pushdown, there are limits on how big the UNION ALL views can be when considering pushdown. For example, if you had a 12-branch UNION ALL view joined with a 16-branch UNION ALL view, that join would not be pushed down since it would create 192 joins. These limits are in a state of revision and hence not specified here.

4.4 GROUP BY pushdown

This internal transformation attempts to push a GROUP BY through a UNION ALL view so that the grouping operation can be applied early on a smaller set of rows. By pushing down a GROUP BY, DB2 can avoid large sorts and can exploit available indexes if they provide the order on the grouping column. GROUP BY pushdown is especially effective when grouping on the partitioning column, but it can be applied in almost all cases. GROUP BY pushdown is only performed in certain circumstances:

- Aggregate functions must be any of MIN, MAX, SUM, COUNT, or AVG.
- The number of remaining branches after partition elimination must be fewer than 36. This limit might be revised in future versions of DB2.

Let's go back to Query 6. After GROUP BY pushdown, the query looks like this:

Query 8:

```
with  s25 as (select s.prod_id, p.prod_desc, g.city, c.channel,
                     sum(s.revenue) as totrev
      from sales_0100 s, products p, geographies g, channel c
      where sales_date >= '01-01-2000' and sales_date <= '29-02-2000' and
            s.prod_id = p.prod_id and
            s.city_id = g.city_id and
            s.channel_id = c.channel_id and
            p.terminated = 'N'
      group by s.prod_id, p.prod_desc, s.city_id, g.city,
               c.channel_id, c.channel),
    s26 as (select s.prod_id, p.prod_desc, g.city, c.channel,
                   sum(s.revenue) as totrev
      from sales_0200 s, products p, geographies g, channel c
      where sales_date >= '01-01-2000' and sales_date <= '29-02-2000' and
            s.prod_id = p.prod_id and
            s.city_id = g.city_id and
            s.channel_id = c.channel_id and
            p.terminated = 'N'
      group by s.prod_id, p.prod_desc, s.city_id, g.city,
```

```
                s.channel_id, c.channel),
    sales2 as (select * from s25
               union all
               select * from s26)
select prod_id, prod_desc, city, channel, sum(totrev) as "Total Revenue"
  from sales2
  group by prod_id, prod_desc, city_id, city, channel_id, channel
```

This transformed Query 8 is shown graphically in Figure 5:

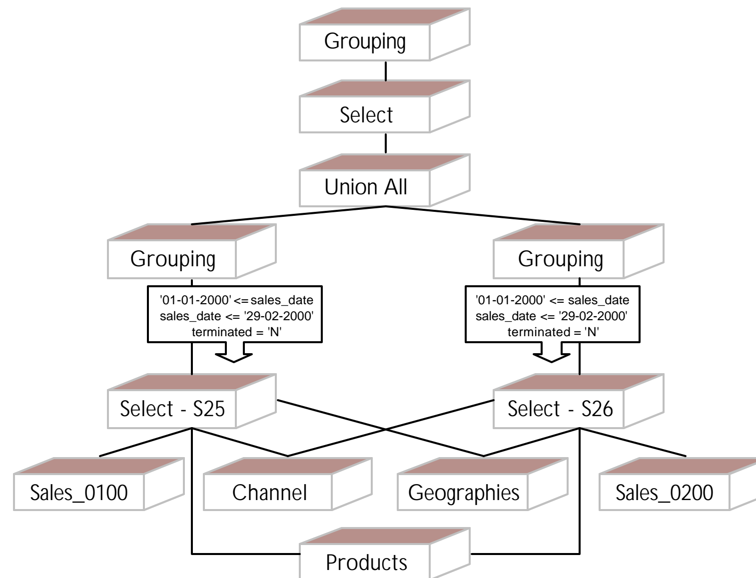

Figure 5. Graphical representation of Query 8

This query is more efficient because the grouping has been done earlier on a smaller set of data before the UNION ALL operation, which means that a smaller set of data must be sorted to perform the grouping. It may not need to be sorted at all if the chosen join method outputs its results in sorted order.

Above the UNION ALL view, there must be another aggregation to combine the results of the lower GROUP BYs. However, this final GROUP BY is very efficient because the rows are partially consolidated before the final aggregation. In this example, above the UNION ALL there are only two rows per group for the top grouping operation to combine, one from each branch of the union.

The drawbacks of GROUP BY pushdown are the same ones for join pushdown: compile time and replication of complex structures. As with join pushdown, there are limits in the optimizer to prevent the pushdown if the GROUP BY expressions are too complex.

4.5 The result of the query rewrite transformation

After all of the above optimizations have been applied, our sample query has been extensively changed. All of the optimizations have to do with performing operations earlier so that later processing works on smaller data sets. The two diagrams below illustrate this. The first diagram, Figure 6, shows the original query, the second, Figure 7, shows the final query as depicted in the previous section. Above each operation are the estimated number of rows that the operation returns.

The diagrams show that each operation has smaller input sets in the rewritten query than it does in the original query. This allows each operation to be more efficient, which in turn allows the entire query to be more efficient.

The optimization level setting within DB2 is a vehicle that allows the user to control the various query transformations and optimization strategies used in the DB2 compiler. As you can probably see by now, there can be a significant amount of processing and memory required to optimize queries that use UNION ALL views. For this reason, most of these optimizations are performed only at optimization level 5 and higher.

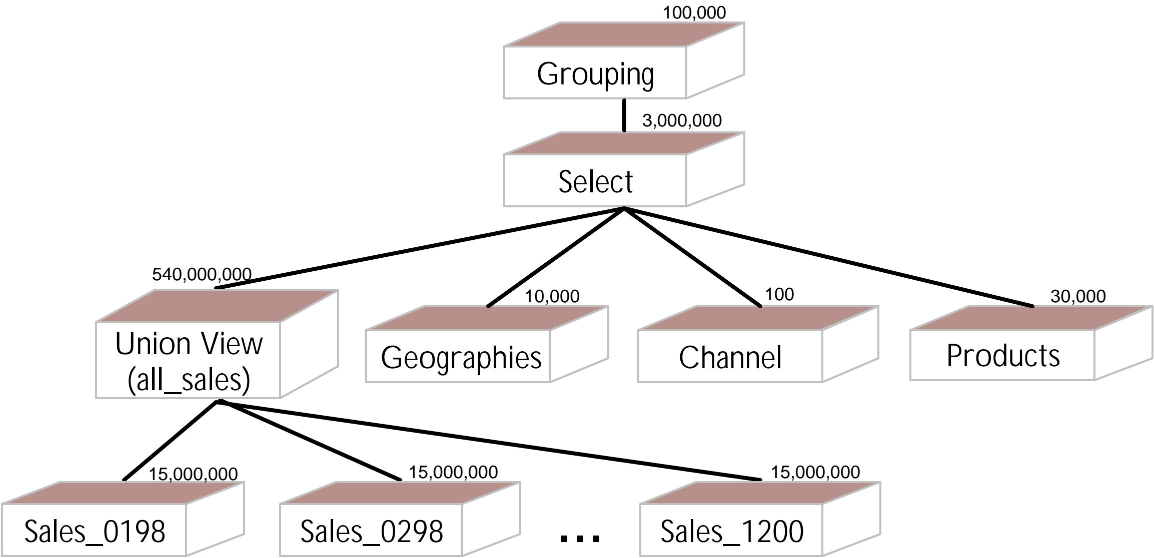

Figure 6. Estimated number of rows in each operation in the original query

After all the optimizations discussed above, the query is executed as follows :

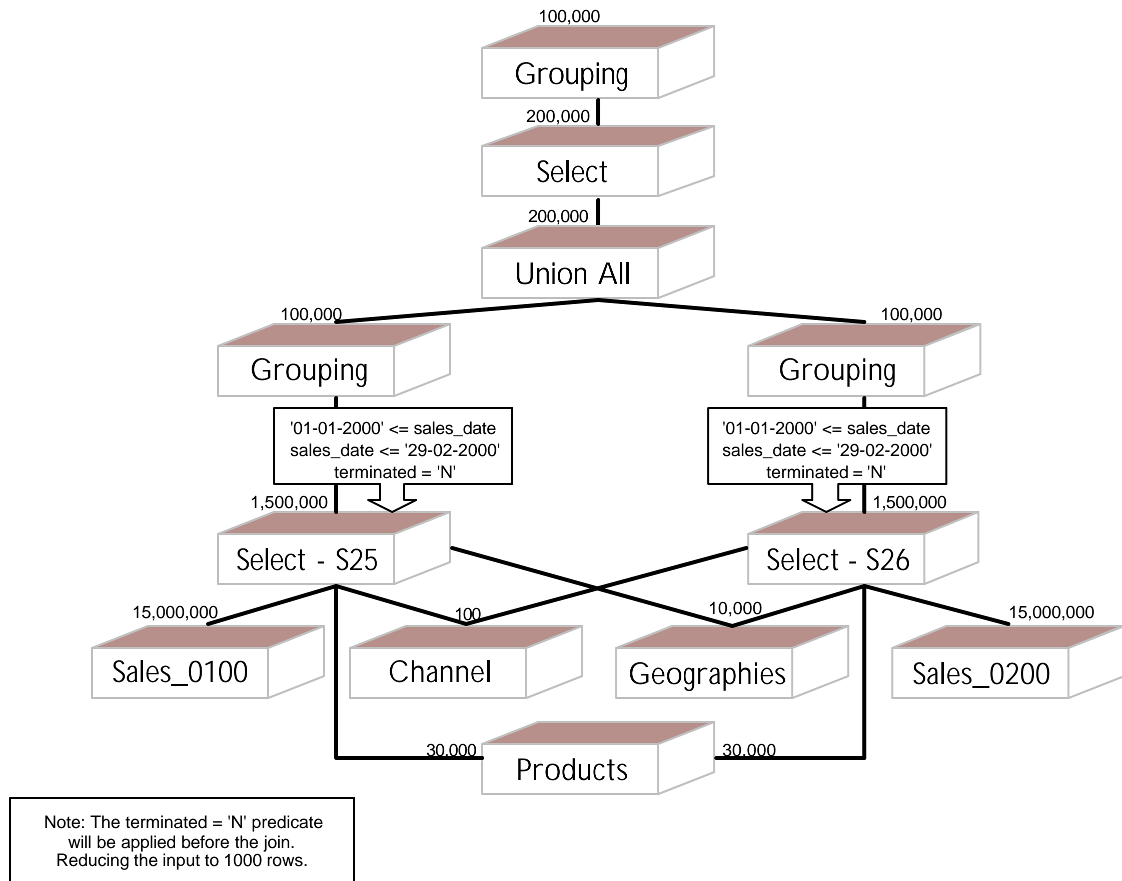

Figure 7. Estimated number of rows in each operation in the optimized query

4.6 Runtime branch elimination with parameter markers

Thus far, we have described branch elimination at compile time. What about runtime branch elimination? DB2 can do some form of runtime branch elimination for parameter markers when equality or range predicates are used in the UNION ALL view. Runtime branch elimination is not handled when there are check constraints. For the curious, the following statement shows the definition of the view *all_sales* needed to allow runtime branch elimination in DB2 Version 7.

```
create view all_sales as
    (
    select * from sales_0198
    where sales_date between '01-01-1998' and '31-01-1998'
      union all
    select * from sales_0298
    where sales_date between '01-02-1998' and '28-02-1998'
      union all
    ...
      union all
    select * from sales_1200
    where sales_date between '01-12-2000' and '31-12-2000'
    );
```

Let's once again go back to our example from Section 4, except that the range predicate is now using host variables.

Query 9:

```
select s.prod_id, p.prod_desc, g.city, c.channel,
       sum(s.revenue) as "Total Revenue"
from products p, geographies g, channel c, all_sales s
where s.prod_id = p.prod_id and
  s.city_id = g.city_id and
  s.channel_id = c.channel_id and
  s.sales_date between :hv_dt1 and :hv_dt2 and
  P.terminated = 'N'
group by s.prod_id, p.prod_desc, s.city_id, g.city, s.channel_id, c.channel
```

All of the optimizations discussed in Section 4 apply here, but branch elimination will not be effective, because the value of *:hv_dt1* and *:hv_dt2* are not known at compile time. After all optimizations are applied, the query looks like this:

Query 10:

```
with p1 as (select prod_id, prod_desc from products
            where terminated = 'N'),
  p2 as (select prod_id, prod_desc from products
         where terminated = 'N'),
  s1 as (select s.prod_id, p.prod_desc, g.city, c.channel,
                sum(s.revenue) as totrev
    from sales_0198 s, p1 p, geographies g, channel c
    where :hv_dt1 <= sales_date and sales_date <= :hv_dt2 and
          sales_date >= '01-01-1998 and sales_date <= '01-31-1998' and
          s.prod_id = p.prod_id and
          s.city_id = g.city_id and
          s.channel_id = c.channel_id
    group by s.prod_id, p.prod_desc, s.city_id, g.city,
             c.channel_id, c.channel),
  ...
  s36 as (select s.prod_id, p.prod_desc, g.city, c.channel,
                 sum(s.revenue) as totrev
    from sales_1200 s, p2 p, geographies g, channel c
    where :hv_dt1 <= sales_date and sales_date <= :hv_dt2 and
      sales_date >= '01-12-2000' and sales_date <= '31-12-2000' and
      s.prod_id = p.prod_id and
      s.city_id = g.city_id and
      s.channel_id = c.channel_id
    group by s.prod_id, p.prod_desc, s.city_id, g.city,
             s.channel_id, c.channel),
  sales2 as (select * from s1
             union all
             select * from s2
             union all
             ...
             select * from s26)
select prod_id, prod_desc, city, channel, sum(totrev) as "Total Revenue"
from sales2
```

```
group by prod_id, prod_desc, city_id, city, channel_id, channel
```

The view predicates are shown in bold. From the predicates of s1, it is easy to derive the following predicates: `:hv_dt1 <= '31-01-1998'` and `'01-01-1998' <= :hv_dt2`.

A limited form of runtime branch elimination is present in DB2 Version 7, which requires you to define your partitioning constraints in the view. In the near future, check constraints will also be handled. These generated predicates will be evaluated at runtime before any table or index accesses occur. If at runtime `:hv_dt1 = '01-02-1998'` and `:hv_dt2 = '31-03-1998'`, then the derived predicate become `'01-02-1998' <= '31-01-1998' and '01-01-1998' <= '31-03-1998'`. The first one is obviously false, and the branch is not accessed even though it was not eliminated at compile time.

Runtime branch elimination can be seen in the optimizer plans even though the elimination doesn't occur until the query is executed. In a typical optimizer plan you might find the following:

```
               0.507407
                NLJOIN
                (   5)
                31.9794
                   1
             /------+-----\
      0.037037           13.7
        TBSCAN           TBSCAN
        (   6)           (   7)
       0.0491778         31.9302
           0                1
          |                 |
          1               137
    TABFNC: SYSIBM      TABLE: SALES
     GENROW             SALES_0198
```

This is an indication of where runtime branch elimination will occur. The left hand branch from table function SYSIBM.GENROW is where the runtime branch elimination occurs. The following will be evaluated first:

```
    7) Residual Predicate
                Relational Operator:              Less Than or Equal (<=)
                Subquery Input Required:    No
                Filter Factor:                    0.333333

                Predicate Text:
                --------------
                (:HV_DT1 <= 31-01-1998)
```

This predicate `:hv_dt1 <= '31-01-1998'`, shown above, is going to be evaluated first, and if it is false, table *sales.sales_0198* will not be accessed.

5. Benefits of UNION ALL views

In addition to the benefit on the query performance, partitioned views also have many other advantages. These benefits include:

- Better control of maintenance window utilities.
- Easier to roll data in and out.
- Ability to leverage different storage media.
- Branch-based performance tuning.

- Schema and data evolution.
- Decreased I/O through branch elimination.
- Increased parallelization.

5.1 Better control of maintenance window utilities

Using a UNION ALL view for large tables various utilities can now be performed on smaller tables. This could be very useful particularly when maintenance windows are small. For example, reorganization of the table, creating indexes and gathering statistics will take less time because it takes less time to complete these utilities on each individual table, and in addition, the process can be parallelized. You can run utilities like RUNSTATS on newly rolled in data that is put in a separate table without having to collect statistics on the other tables involved in the UNION ALL. When rolling in of new data into a single large table, certain statistics like COLCARD, the number of distinct values in a column, cannot be computed without looking at the values in the existing data. A value in the newly loaded data may or may not exist in the table and so COLCARD cannot be computed by simply looking at the new data.

5.2 Easier to roll data in and out

Let's consider our business scenario again. Every quarter, the system administrator has to roll out the data of the three oldest months to make room for the data for the new quarter.

If all data is in a single table, the rows are deleted. In the example given above, there would be 540 million rows in the sales tables. Rolling out a single month involves deleting 15 million rows, which might be unworkable due to the logging time required. Disabling logging might not be an advisable solution.

The original view in our example above referenced data from January 1998 to December 2000. To roll in the data for January 2001 and roll out the data pertaining to January 1998:

- Create and populate table *sales_0101* with data for January 2001.
- Drop view *all_sales*.
- Recreate the view *all_sales* with the following definition adding the reference to *sales_0101* and removing the reference to *sales_0198*:

```
create view all_sales as
    (select * from sales_0298
    where sales_date between '01-02-1998' and '28-02-1998'
      union all
    select * from sales_0398
    where sales_date between '01-03-1998' and '31-03-1998'
      union all
    ...
      union all
    select * from sales_1200
    where sales_date between '01-12-2000' and '31-12-2000'
      union all
    select * from sales_0101
    where sales_date between '01-01-2001' and '31-01-2001'
    );
```

- After the view has been redefined, the table *sales_0198* can be archived or deleted depending on business needs.

If the granularity of the partitions is coarser than the data rolled in or out, it might seem that it is not as simple as just dropping a table and adding a new one to the view. However, by using WHERE predicates in the UNION ALL view, it is possible. For example, if the UNION ALL view was over tables that contained data for a quarter (3 months) and we needed to roll in and roll out a month's worth of data, this needs to be done a little differently that what is described above. A suitable efficient way could easily be devised.

5.3 Ability to leverage different storage media

One of the key applications that differentiates the UNION ALL view from all other partition methods, including Oracle's range partitioning, is the ability to effectively exploit different types of media using products such as Tivoli® Hierarchical Storage Manager (HSM). The relative independence between the individual partitions allows the tables to be defined in separate table spaces. This allows some of the infrequently queried partitions to be stored on tape or other slow media.

The UNION ALL view allows the archived data to be easily accessed. With the optimizations shown in Section 4, only archival media that contained relevant data would need to be accessed. Because this archived data would be used very infrequently, it may be desirable to store it on slow archive media such as tapes.

Such a solution could be invaluable to an organization that has a large amount of data that is associated with a single relation. An excellent description of this solution is available in a white paper from the IBM Teraplex centre at the following web site: http://www.ibm.com/software/data/pubs/papers/hierstorage.

Incidentally, there are some useful scripts in the paper for defining the UNION ALL views.

5.4 More granular performance tuning

With a single table solution, all performance tuning affects the entire table. For example, indexes are uniformly defined across all the data, and statistics are updated at the same time.

With a UNION ALL view, however, each table from a branch of the UNION can have independent indexes. In addition, you can collect and update statistics only for those branches in which the data has changed. Highly used tables may have more indexes defined or be kept on some faster media than infrequently used tables. This provides incredible flexibility to the system administrator when dealing with performance and space issues.

To reduce the number of tables managed by the UNION ALL view, you also have the flexibility to skew the data. In other words, you can create large tables that cover the older data and relatively smaller tables that cover the latest data. Because each UNION ALL branch is optimized independently, DB2 might use different access plans to access larger tables compared to the smaller ones.

5.5 Easier to modify the schema and the data

The evolution of the data to suit business needs is easily accomplished with a UNION ALL view.
For example, assume that in February 2001, due to an organizational change, the sales department was divided into three departments. Now it is necessary to break down the data in column *revenue* separately for the three departments.

The table for January 2001 was defined as :

```
sales_0101(    sales_date   date  not null,
    prod_id    integer,
    city_id    integer,
    channel_id   integer,
    revenue  decimal(20,2))
```

The table for February can be defined as follows:

```
sales_0201(  sales_date   date  not null,
    Prod_id    integer,
    city_id    integer,
    channel_id   integer,
    revenue_dept1   decimal(20,2),
    revenue_dept2   decimal(20,2),
    revenue_dept3   decimal(20,2))
```

With a single table solution, the challenge is how to fit data into the new schema. You would have to redefine the entire table to include the new columns. There is also a problem of what to put in those columns for the historical data.

With a UNION ALL view, the answer is much easier—depending on the application change requirements, you can redefine the view in one of the following two ways.

Solution 1: *all_sales* view schema remains the same with possibly no change to the applications:

```
 create view all_sales as
 ( select sales_date, prod_id, city_id, channel_id, revenue from sales_0398
  union all
select sales_date, prod_id, city_id, channel_id, revenue from sales_0498
  union all
...
  union all
select sales_date, prod_id, city_id, channel_id,
  revenue_dept1+revenue_dept2+revenue_dept3
  from sales_0201
);
```

Solution 2: *all_sales* view schema incorporates the new columns:

```
 create view all_sales (sales_date, prod_id, city_id, channel_id, revenue,
                        revenue_dept1, revenue_dept2, revenue_dept3)
 as
 ( select sales_date, prod_id, city_id, channel_id, revenue, NULL, NULL, NULL
     from sales_0398
  union all
  select sales_date, prod_id, city_id, channel_id, revenue, NULL, NULL, NULL
     from sales_0498
  union all
...
  union all
  select sales_date, prod_id, city_id, channel_id,
         revenue_dept1+revenue_dept2+revenue_dept3,
```

```
            revenue_dept1, revenue_dept2, revenue_dept3
      from sales_0201
    );
```

The schema has now evolved to suit the new requirements. Initially, it may be beneficial to stay with the old view schema. Once there is some significant amount of data using the new schema, you could then switch to the new view where the schema has changed. This also gives time for any applications that depend on the view to be updated.

5.6 Decreased I/O costs

In Section 4.2, we described how the optimizer can eliminate redundant branches for queries on UNION ALL views. It is certainly an important benefit to reduce the number of branches that must be accessed, but another important benefit not to be overlooked is the ability to reduce I/O for other operations.

To explain this, assume that the business must compare the sales performance between two quarters, for example 1Q 2001 versus 1Q 2000.

In the single table solution that has a large amount of historical sales data, the chosen access plan would likely involve performing an index scan on the sales_date column to pick out the rows for each quarter and then accessing the table to fetch the remainder of the data. Given that the table has 540 million rows and that only 90 million are required for the query, the index scan would be discarding most of the index entries it finds.

On the other hand, with the UNION ALL view, branch elimination would have reduced the number of tables that are relevant to six. The way the tables are defined makes it possible to know that all of the required data is contained in those tables. The optimizer can now choose to do a sequential table scan on those tables, avoiding the index altogether. This can save a significant amount of I/O cost.

5.7 Increased query parallelization

In addition to reducing I/O cost, a UNION ALL view leads to increased parallelization in DB2 EEE. In the example from Section 5.6, the query compares the sales performance for two quarters 1Q 2001 versus 1Q 2000. With branch elimination, all unnecessary I/O was eliminated. However, the two table scans still execute serially. IBM DB2 EEE can parallelize query processing across multiple “nodes”. Each node executes operations in parallel with the other nodes and uses a “hash” partitioning scheme to distribute data among the nodes.

The hash partitioning used in DB2 EEE integrates well with using a UNION ALL view on large quantities of data. A UNION ALL view can define a mix of range partitioning and hash partitioning; this allows the *view* data to be partitioned into ranges, and allows the *base* tables to be distributed across the nodes of the database. The effect of this two-level partitioning in EEE is that queries that access multiple tables can be executed in parallel.

If our example query had been executed on a EEE database and if the base tables had been distributed to different nodes, the scans of the six tables could be executed in parallel and only need to be recombined at the end to provide the output to the user. All of the optimizations that the DB2 query rewrite performs can benefit by this. GROUP BY and join pushdown allow these operations to be performed in parallel. It is not always possible for DB2 to execute the UNION ALL branches themselves in parallel.

5.8 Optimizing UNION ALL views in a federated environment

The UNION ALL view is commonly used in a federated environment where branches of the UNION ALL view have tables in remote databases. An appropriate paper “Building Federated Systems with Relational Connect and Database Views” relevant to the UNION ALL usage in the federated environment is available at http://www7b.boulder.ibm.com/dmdd/library/techarticle/rutledge/0112rutledge.pdf. This paper describes a proof-of-concept design for a federated database scenario connecting a legacy bank system in DB2 and a legacy brokerage system in Oracle.

6. Limitations of using UNION ALL views

There are some important limitations involved with using UNION ALL views. Although some of these limitations are planned to be addressed in future versions of DB2, consider them before choosing UNION ALL views for solving the problem of massive data quantities. These limitations include:

- Risk of view materialization with complex queries.
- Problems guaranteeing uniqueness across the tables in the view.
- Restriction on inserting into a UNION ALL view.
- Limitations on the number of branches when creating a UNION ALL view.
- Increased compile time and memory usage.

6.1 Risk of view materialization with complex queries

Depending on the complexity of the query or the definition of the view, DB2 may not be able to perform predicate pushdown or join pushdown. In such situations, the entire view may have to be materialized. If such a situation leads to unacceptable performance, it may be necessary to simplify the query. This may be particularly significant if there is more than one reference to UNION ALL views within the query.

6.2 Problems guaranteeing uniqueness across the tables in the view

One limitation of the UNION ALL view approach is the inability to create globally unique indexes across tables. In our sales data, assuming that each transaction was to be given a unique number, an index cannot be used to ensure global uniqueness across the branches of the UNION ALL view. You need another method to ensure this. One solution is to use a SEQUENCE value, which is guaranteed to be unique across the database. Details of this feature are in the DB2 *SQL Reference*. Note that this feature is not supported in the partitioned environment of DB2 Version 7.

Not having globally unique indexes can also lead to performance problems in specific cases. Given a unique ID for each transaction, if we wanted a list of all transactions in the order of the unique ID, a sort would need to be performed. With a globally unique index, the transaction could be read out of the index in order. In addition, such an ID column cannot be supported in a referential integrity relationship where it is the primary key.

6.3 Restriction on inserting into a UNION ALL view

In DB2 Version 7, you cannot insert through a UNION ALL view. Instead, inserts must be performed directly into the individual tables that are used in the UNION ALL view. However, this is a temporary limitation. Work is in progress for future versions to allow the data to be loaded directly through the *all_sales* view and to let DB2 insert the data into the correct monthly sales table. DB2 will determine the proper table by the definition of check constraints on the table. Updates to the UNION ALL partitioning column is assumed to be infrequently done if at all. If the UNION ALL view has view predicates to partition the view and no check constraints, we could use triggers to carry out the update. The triggers, if appropriate, could DELETE the row from the table and INSERT it into another table that corresponds to the new value of the partitioning column. With check constaints, before the triggers are fired, the initial UPDATE could violate the check constraint and fail. In this case, the application might have to split the UPDATE into a DELETE and an INSERT.

6.4 Limitations on the number of branches in a UNION ALL view

When a CREATE VIEW statement is executed, DB2 has work to do to validate the syntax and semantics of the view. This involves reading all the information associated with each table and resolving the columns types just as is done when compiling the statement used to define the view. Each branch of the UNION ALL view references an independent table. It could be that the tables involved have the same number of columns and same types and probably even similar statistics. However, DB2 considers these as independent tables, and it consumes memory to store this information during compilation. Therefore, there are limits to the number of tables that can be defined in the view. A simple test shows that this is somewhere in the order of 400 tables. This is dependent on the complexity of the tables and the extent to which statistics are collected.

6.5 Increased compile time and memory usage

For a UNION ALL view with a large number of branches, substantial work is required to do the optimizations described in Section 4. In addition, the optimizer looks at each branch and does an independent optimization to look for the best access plan for each branch. This implies that the compile time is magnified depending on the complexity of the query after all the transformations are done. With joins pushed down through the UNION, the memory requirements can quickly multiply. We recommend increasing the size of the STMTHEAP database configuration parameter, which is associated with the memory allocated for use by the DB2 compiler.

If the branches of the UNION ALL view are not eliminated, there is also an increased memory requirement when building and executing the plan chosen by the optimizer. Other than the STMTHEAP, it may be necessary to increase the APPLHEAPSZ database configuration parameter as well. A simple test showed that, by applying the maximum of those two configuration parameters, there are limits on the size of the UNION ALL view that can be used.

The general recommendation is to limit the UNION ALL branches to a reasonable size; with DB2 V7.2, the recommendation is to keep it less than 36, and smaller if compile time is an issue.

Theoretical limits on the number of branches involved in a UNION ALL view are between 100 and 200 depending on the other components of any query accessing the view. Using a UNION ALL view where the number of branches is approaching these limits will lead to sub-optimal performance in the execution of the query.

6.6 Future Enhancements

UNION ALL views are being greatly improved to make this a viable solution for dealing with large amounts of data. Improvements in predicate analysis will be done to better handle IN predicates. There will also be capabilities to INSERT through a UNION ALL view. Over and above the transformations described above, additional optimizations will be considered to reduce the quantity of data transferred from remote sources in a federated environment. As always, effort is being put in to improve compile time. For UNION ALL views, this is a an important task considering the complexity after the various pushdown transformations.

8. Conclusion

A divide and conquer UNION ALL view approach has been discussed in this paper and has been shown to be a solution worth considering to :

- Overcome capacity issues.
- Address issues with shorter maintenance windows.
- Improve execution time.

DB2 has a very powerful query rewrite component as part of the DB2 SQL compiler that allows a simple UNION ALL view to be used as an effective solution towards resolving some of these issues. This solution may not be a panacea for all such problems and does have some limitations. However, the information provided in this paper is intended to help you determine if the approach can work for you and to help you make good design decisions in the process.

Notices, Trademarks, Service Marks and Disclaimers

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this document should not be interpreted as such.

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries: DB2, DB2 Universal Database, IBM, Tivoli.

Other company, product or service names may be the trademarks or service marks of others.

The furnishing of this document does not imply giving license to any IBM patents.

References in this document to IBM products, Programs, or Services do not imply that IBM intends to make these available in all countries in which IBM operates.